Java-mis

Java Enum
Java Lamnda

Contents

Java ?

List-forEach()

```
List<String> items = new ArrayList<>();
items.add("A");
items.add("B");
items.add("C");
items.add("D");
items.add("E");

//lambda
//Output : A,B,C,D,E
items.forEach(item->System.out.println(item));
```

Ez miért m?ködik?

A List a java.lang.Iterale osztály leszármazottja. Ebben van egy default metódus implementáció a forEach-re.

```
package java.lang;

public interface Iterable<T> {
...
    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
...
```

Ez a **Consumer<? super T>** generikus interfész implementációt várja paraméterként, ahol a T a List-ben lév? lista elemek típusa. A T listaelemek bármilyen leszármazottját elfogadja. Láthatjuk az implementációból hogy a forEach metódus az összes listaelemen végigmegy, és minden egyes listaelemre meghívja a Consumer<T> implementáció **accept()** metódusát. A lista elemeket a saját osztály példányának a példány változójából szedi (vagyis ha van egy ArrayList típusú objektumunk, akkor az maga tárolja a lista elemeket egy bels? változójában). A Consumer interfész implementáción múlik, hogy mit fog csinálni az adott listaelemmel.

A Consumer implementáció (Consumer<? super T> action) csak egy cs?, amibe felülr?l be tudunk dobni egy T elemet és kidob alul egy R elemet (jelen esetben nincs R elem, mert void típusú a Funkcionális függvény). Tehát az implementációnak nincs állapota. Kap egy inputot és generál bel?le egy outputot.

A Consumer interfészben az accept az egyetlen absztrakt metódus, tehát a Consumer egy funkcionális interfész, ahogy ez az annotációból is látszik:

```
package java.util.function;
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

Note
A **@FunctionalInterface** annotáció hatására a fordító ellen?rzi, hogy az adott interfész valóban megfelel e a funkcionális interfészekkel támasztott követelményeknek.

- Csak interfészre kerülhet ilyen annotáció
- Az interfésznek pontosan 1 darab absztrakt metódusa van (vagyis implementálandó)

Nézzünk egy lehetséges implementációt:

```
public class MyConstumer implements Consumer<String>{
@Override
public void accept(String t) {
System.out.println(t);
}
```

```
}
```

A MyConsumer osztály implementálja a Consumer interfész egyetlen metódusát, az accept-et, egy String-et vár inputba és nem ad vissza semmit.

Ennek az implementációs osztálynak ez lenne egy lehetséges felhasználása:

```
public static void main(String[] args) {
List<String> items = new ArrayList<>();
items.add("A");
items.add("B");
items.add("C");
items.add("D");
items.add("E");

Consumer<String> myConsumer = new MyConstumer();
items.forEach(myConsumer);
}
```

Azonban Java8-tól kezdve bevezették a ? kifejezéseket, mivel név nélküli inline függvény implementációkat készíthetünk, ezek tulajdonképpen input és output paraméterrel rendelkez? kód blokkok az alábbi szintaxissal (alább három kül. példa)

```
parameter -> expression
(parameter1, parameter2) -> expression
(parameter1, parameter2) -> { code block }
```

Warning
Ezek a paraméterek nem a "küls?", funkcionális interfészt implementáló java osztályt meghívó metódus bemeneti paraméterei, hanem bels? változok. A **forEach** esetén pl ezek a lista elemek

Az el?z? példa Lambda alakja a következ? lenne:

```
Consumer<String> myConsumer = item -> System.out.println(item);
items.forEach(myConsumer);
```

És a lambda kifejezést írhatjuk kapásból a forEach argumentumába hogy rövidítsünk:

```
items.forEach(item->System.out.println(item));
```

Note
Az item kívülr?l nézve nem értelmezhet?, szemben egy "normál" függvény hívással, ahol kintr?l adunk át neki paramétereket. Tehát ez NEM egy küls? paraméter. Értelmet csak a forEach belsejében nyer. A lambad kifejezés csak egy állapot nélküli transzformációs függvény, ami inputnak megkapja az '*item* nev? változót és csinál vele valamit:

A Lambda kifejezést kicserélhetjük method reference-re is:

```
items.forEach(System.out::print);
```

Ez egy statikus metódus referencia, amivel azt mondjuk meg, hogy a Consumer implementációja a System.out.print(T) metódust fogja meghívni. A lényeg, hogy egy argumentumos legyen a megadott metódus és fel tudjon dolgozni T lista típusú elemet.

Példa 2: saját implementáció

Az alábbi osztálynak van egy darab osztály változója (variable1) és van egy darab metódusa: processVariable. Ez a metódus egy funkcionális interfészt vár input paraméterként.

```
package hu.otp.auth.loginHelper.util;
import java.util.function.Function;

public class MyLambadClass {

private String variable1;

public MyLambadClass(String a) {
this.variable1 = a;
}

public void processVariable(Function<String, Integer> fn) {

Integer length = fn.apply(variable1);
System.out.println(length.toString());
}
}
```

A processVariable belsejében meghívja a funkcionális interfész **apply()** metódusát. (Definiálhattunk volna magunknak saját funkcionális interfész típust, mi most itt egy gyári típust használunk).

```
package java.util.function;
```

```
import java.util.Objects;

@FunctionalInterface
public interface Function<T, R> {
    R apply(T var1);   <<<<<<<<<<<----------------------------------------------------ez itt a lényeg!!!

    default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
        Objects.requireNonNull(before);
        return (v) -> {
            return this.apply(before.apply(v));
        };
    }

    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (t) -> {
            return after.apply(this.apply(t));
        };
    }

    static <T> Function<T, T> identity() {
        return (t) -> {
            return t;
        };
    }
}
```

Láthatjuk, hogy egy implementálandó metódusa van a "Function" interfésznek, ami T típust vár és R típus ad vissza. Tehát ez egy generikus interfész, és bármi is legyen az implementáció, ott majd meg kell mondani, hogy a pl T=String és R=Intger. De az hogy a Function az egy generikus interfész (generic) annak semmi köze ahhoz hogy ? egy Funkcionális interfész.

Hagyományos implementáció

A Function interfésznek elkészíthetjük a saját implementációnkat is:

```
class MyFunctionImpl implements Function<String, Integer> {
        @Override
        public Integer apply(String s) {
            return Integer.valueOf(s.length());
        }
    }
```

Az el?bb láthattunk, hogy egy metódust kell kötelez?en definiálni a Function interfészen, ez az apply. Az implementáció els? sorában kikötöttük, hogy T és R String és Integer lesz. Tehát az apply() implementációnknak S-t kell kapnia paraméterként és Integer-t kell visszaadnia. Ez itt teljesül. Nem csinál mást mint a kapott String hosszát visszaadja.

Ahhoz hogy ezt használni tudjuk, példányosítani kell, és át kell adni processVariable() metódusnak, ami egy példányosított funkcionális interfészt implementáló osztályt vár. Vagyis egy olyan objektumot, ami egy olyan osztályból készült, ami implementálja a funkcionális interfész metódusát.

```
MyLambadClass myLambadClass = new MyLambadClass("adam");

        MyFunctionImpl myFunction = new MyFunctionImpl();

        myLambadClass.processVariable(myFunction);
```

A processVariable() implementációját mi készítettük, tudjuk hogy nem csinál mást, mint meghívja a myFunction.apply() metódust.

? implementáció

Nézzünk az alábbi lehetséges ? implementációt (inline). Az apply-nak átadja a saját osztály változóját, majd az apply által visszaadott Integer-t kiírja a sysout-ra. Az hogy az apply belsejében hogy áll el? az Ingerer és hogy mit csinál a bemen? String paraméterrel teljes egészében az apply implementációra van bízva.
Nézzünk meg egy lehetséges implementációt:

```
public static void main(String[] args) {

MyLambadClass myLambadClass = new MyLambadClass("adam");
myLambadClass.processVariable(var1 -> { return Integer.valueOf(var1.length()); });

}
```

Ebb?l az apply lambad kifejezéssel történ? inline implementációja az alábbi:

```
var1 -> { return Integer.valueOf(var1.length()); }
```

Ami nem csinál mást, mint visszaadja a kapott string hosszát. Kívülr?l nézve a var1 nem értelmezhet?, az mindig a lamdát futtató osztály egy osztály változója, kívülr?l nem megadható.

Warning
Itt a 'var1' nem kívülr?l jöv?, a metódus meghívásakor el?állt paraméter! Ez a 'MyLambadClass' bels? változója, vagyis ide kívülr?l nem tudunk változót betolni a metódus meghívásakor

Tehát, ahogy ezt majd látni fogjuk a CompletionSage-nél, a 'FunctionalInterface'-t futtató osztályt kell el?re feltölteni minden olyan változóval, amire szükség van a lamda kifejezés futtatására. A fenti példákban ez egyrészt a 'MyLambadClass', vagy az els? példában a ArrayList osztályok.

- A 'MyLambadClass' konstruktorában adtuk át azt a string-et, amit a 'processVariable' feldolgoz, attól függetlenül, hogy milyen lamda kifejezéssel implementáljuk a funkcionális interfészét.
- Az ArrayList pedig bels? változóiban tárolja

Példa 3: CompletionStage.thenCompose()

Mi is ez?

- **CompletionStage Interface**:
  - ezek egymás után futtatott stage-ek, ezért hívják "Comletion"-nek. Mikor az egyik stage véget ér, potenciálisan elindít egy másik "CompletionStage"-et és így tovább, egymásba láncolva. Minden metódusa egy másik CompletionStage-et ad vissza. De ez csak egy interfész, vmilyen implementációját kell használni.
  - java.util.concurrent.CompletionStage<T> interface represents a commutation task (either synchronous or asynchronous). As all methods declared in this interface return an instance of CompletionStage itself, multiple CompletionStages can be chained together in different ways to complete a group of tasks.
- **CompletableFuture<T> implements CompletionStage<T> and Future<T>**: The static factory methods in this class are the starting points for executing tasks.

Nézzünk egy példát a CompletionStage láncHívására. A CompletionStage minden metódusa CompletionStage-el tér vissza, köztük a thenComponse() is, ezért hívható láncban.

```
method1().thenCompose(var1 -> method2(var1))
         .thenCompose(this::method3))
         .thenCompose(var2 -> { System.out.println(var2.printMe());
                                System.out.println("That was it"); return new CompletionStage(..) ;
                               } )
         .whenComplete((result, error) -> { ...}
```

A láncot indító metódusnak értelem szer?en egy **CompletionStage** példánnyal kell visszatérnie.

A **thenCompose** szignatúrája az alábbi:

```
<U> LogCompletionStage<U> thenCompose(Function<? super T, ? extends CompletionStage<U>> var1);
```

Láthatjuk, hogy egy **Function** interfészt vár paraméterül. A 'Function' interfésze nem más mint egy "gyári" funkcionális interfész típus (olyan mint az el?z? példába a 'Consumer' interfész volt). A funkcionális interfészekkel szemben támasztott követelmény, hogy egy nem implementált metódusuk legyen, ami ebben az esetben az **apply()**. A **T** az input típusa, míg **R** a visszatérési típus.

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T var1);
    ...
}
```

Tehát a **thenCompose(..)** vár egy **Function** interfész implementációt és belül meg fogja hívni az **apply(..)** metódusát, aminek át fog adni egy T típusú változót, és R típusú változót fog visszavárni visszatérési értékként. Nézzük meg a **thenCompose** implementációját:

```
public <U> CompletableFuture<U> thenCompose(
        Function<? super T, ? extends CompletionStage<U>> fn) {
        return uniComposeStage(null, fn);
    }

    private <V> CompletableFuture<V> uniComposeStage(
        Executor e, Function<? super T, ? extends CompletionStage<V>> f) {
        if (f == null) throw new NullPointerException();
        CompletableFuture<V> d = newIncompleteFuture();
        Object r, s; Throwable x;
        if ((r = result) == null)
            unipush(new UniCompose<T,V>(e, d, this, f));
        else if (e == null) {
            if (r instanceof AltResult) {
                if ((x = ((AltResult)r).ex) != null) {
                    d.result = encodeThrowable(x, r);
                    return d;
                }
                r = null;
            }
            try {
                @SuppressWarnings("unchecked") T t = (T) r;
                CompletableFuture<V> g = f.apply(t).toCompletableFuture();
        ....
    }
```

Anélkül hogy nagyon belemennénk a részletekbe, a lényeg itt is az mint a 2. példában. A CompletableFuture példánynak vannak példány változói. Fogja a T típusú változóját és meghívja vele az **apply(..)** metódust, és visszavár egy CompletionStage leszármazottat. Mivel a CompletionStage hívások láncba f?zhet?ek, a hívások egymás után hajtódnak végre. A fenti kódrészlet egy kis varázslással végrehajtja és el?veszi a láncban az el?z? CompletionStage<R> értékét és ebb?l az R-t adja oda az apply(...) metódusnak, hogy végezze el rajta a szükséges m?veleteket.

Az apply(..) metódus inline implementációja tehát mindig egy CompletionStage<R> példánnyal kell hogy visszatérjen, ahonnan az R-t fogja megkapni a következ? thenCompose hívás inline implementációja T-ként. Pszeudokóddal leírva ez így néz ki:

```
...
   .thenCompose(var1 -> {  ...
                        //Create a future with string inside
                        new CompletableFuture<String> future = new CompletableFuture<String>();
                        future.complete("adam");
                        return future;
                    })
   .thenCompose(var1 -> {
                         System.out.println(var1); //will print "adam"
                        ...
                        })
   ...
```

Az els? thenCompose(..) vissza fog térni egy CompletableFuture-el, amiben egy String-et tárol el. A második thenCompose(..) hívás végre fogja hajtani az els?t, majd annak az eredményéb?l ki fogja nyerni az "adam" Stringet és azzal fogja meghívni a Funkcionális interfész inline implementációt vagyis olyan mint ha ezt csinálná:

```
fn.apply("adam");
```

Note
Tehát az apply(..) inline implementációja csak egy cs?, amibe a thenCompose() bedob egy változót és visszavár egy végeredményt. A változó itt is a saját példány változója (történetesen az el?z? futás eredménye)

Rövidítés

Na de mit jelent ez a sor:

```
.thenCompose(this::method3)
```

Ez csak egy rövidítés. Egy statikus metódus referencia. Ahelyett hogy megadnánk egy inline implementációt, azt mondjuk meg, hogy ez a függvény végzi el azt a feladatot, amit az inline implementációnk végzett volna el.

```
<Melyik osztály>::<melyik metódusa>
```

Ezt akkor lehet alkalmazni, ha amúgy is csak csak metódus hívást írtunk volna a body-ba:

```
.thenCompose(var2 -> method3(var2))
```

Java Method Reference and Constructor Reference

https://www.amitph.com/java-method-and-constructor-reference/

Method Reference

Constructor Reference

Stream

https://stackify.com/streams-guide-java-8/ ..TODO..